Electronic Colloquium on Computational Complexity, Report No. 99 (2013)

Local reductions

Hamid Jahanjou* Eric Miles* Emanuele Viola*

July 6, 2013

Abstract

We reduce non-deterministic time $T \geq 2^n$ to a 3SAT instance ϕ of size $|\phi| = T \cdot \log^{O(1)} T$ such that there is an explicit circuit C that on input an index i of $\log |\phi|$ bits outputs the ith clause, and each output bit of C depends on $O(1)$ inputs bits. The previous best result was C in NC^1. Even in the simpler setting of $|\phi| = \mathrm{poly}(T)$ the previous best result was C in AC^0.

More generally, for any time $T \geq n$ and parameter $r \leq n$ we obtain $\log_2 |\phi| = \max(\log T, n/r) + O(\log n) + O(\log \log T)$ and each output bit of C is a decision tree of depth $O(\log r)$.

As an application, we simplify the proof of Williams' ACC^0 lower bound, and tighten his connection between satisfiability algorithms and lower bounds.

*Supported by NSF grant CCF-0845003. Email: {hamid,enmiles,viola}@ccs.neu.edu

ISSN 1433-8092

1 Introduction

The efficient reduction of arbitrary non-deterministic computation to 3SAT is a fundamental result with widespread applications. For many of these, two aspects of the efficiency of the reduction are at premium. The first is the length of the 3SAT instance. A sequence of works shows how to reduce non-deterministic time-T computation to a 3SAT instance ϕ of quasilinear size $|\phi| = \tilde{O}(T) := T \log^{O(1)} T$ [HS66, Sch78, PF79, Coo88, GS89, Rob91]. This has been extended to PCP reductions [BSGH+05, Mie09, BSCGT13, BSCGT12].

The second aspect is the computational complexity of producing the 3SAT instance ϕ given a machine M, an input $x \in \{0,1\}^n$, and a time bound $T = T(n) \geq n$. Consider ϕ of size $|\phi| \geq T^2$. It is well-known that ϕ is computable by circuits of size poly(T), and that these circuits may be taken even from the restricted class AC^0 of unbounded fan-in, constant-depth circuits consisting of And, Or, and Not gates. In fact, [AAI+01] improve this to NC^0 circuits, i.e., local maps where each output bit depends on a constant number of input bits. For many applications one however needs the construction of ϕ to be *explicit*: the ith clause of ϕ should be computable with resources $\mathrm{poly}(|i|) = \mathrm{poly}\log|\phi| = \mathrm{poly}\log T$ given $i \leq |\phi|$ and $x \in \{0,1\}^n$. When $T = \mathrm{poly}(n)$ this is indeed possible by a simple algorithm running in time $\mathrm{poly}(|i|)$ with random access to x. When $T \geq 2^n$ the size of the index i is more than that of x, and one can obtain a circuit of size $\mathrm{poly}(|i|)$. This in particular yields that the explicit versions of NP-complete problems are complete for NEXP. (Such problems are sometimes called succinct.) Arora, Steurer, and Wigderson [ASW09] push this further and note that one may get an AC^0-explicit reduction: the circuit mapping i and x (both of length $\Theta(n)$) to the ith clause of ϕ is AC^0 of size $\mathrm{poly}(|i|)$. This implies that, unless EXP = NEXP, standard NP-complete graph problems cannot be solved in time $\mathrm{poly}(2^n)$ on graphs of size 2^n that are described by AC^0 circuits of size $\mathrm{poly}(n)$.

Interestingly, applications to unconditional complexity lower bounds rely on reductions that simultaneously optimize both aspects of the reduction. For example, the time-space tradeoffs for SAT need to reduce non-deterministic time T to a 3SAT instance ϕ of quasilinear size $\tilde{O}(T)$ such that the ith clause is computable in time $\mathrm{poly}(|i|) = \mathrm{poly}\log|\phi|$, see e.g. [FLvMV05] or Van Melkebeek's survey [vM06]. More recently, the importance of optimizing both aspects of the reduction is brought to the forefront by Williams' approach to obtain lower bounds by satisfiability algorithms that improve over brute-force search by a super-polynomial factor [Wil10, Wil11b, Wil11a, SW12, Wil13]. To obtain lower bounds against a circuit class C using this technique, one needs a reduction of non-deterministic time $T = 2^n$ to a 3SAT instance of size $\tilde{O}(T)$ whose clauses are computable by circuits in the class C of size $\mathrm{poly}(n)$. For example, for the ACC^0 lower bounds [Wil11b, Wil13] one needs to compute them in ACC^0. However it has seemed "hard (perhaps impossible)" [Wil11b] to compute the clauses with such restricted resources.

Two workarounds have been devised [Wil11b, SW12]. Both exploit the fact that, under an assumption such as $\mathrm{P} \subseteq \mathrm{ACC}^0$, non-constructively there does exist such an efficient circuit computing clauses; the only problem is constructing it. They accomplish the latter by guessing-and-verifying it [Wil11b], or by brute-forcing it [SW12] (cf. [AK10]).

$$\boxed{\text{local} = \mathrm{NC}^0 = \mathrm{DT}(O(1)) \subsetneq \mathrm{DT}(O(\log n)) \subsetneq \mathrm{DNF} \cap \mathrm{CNF} \subsetneq \mathrm{AC}^0 \subsetneq \mathrm{NC}^1.}$$

Figure 1: Inclusion between poly(n)-size circuit classes. DT(d) is for "depth-d decision tree."

1.1 Our results

We show that, in fact, it is possible to reduce non-deterministic computation of time $T \geq 2^n$ to a 3SAT formula ϕ of quasilinear size $|\phi| = \tilde{O}(T)$ such that given an index of $\ell = \log |\phi|$ bits to a clause, one can compute (each bit of) the clause by looking at a constant number of bits of the index. Such maps are also known as local, NC^0, or junta. More generally our results give a trade-off between decision-tree depth and $|\phi|$. The results apply to any time bound T, paying an inevitable loss in $|x| = n$ for T close to n.

Theorem 1 (Local reductions)**.** *Let M be an algorithm running in time $T = T(n) \geq n$ on inputs of the form (x, y) where $|x| = n$. Given $x \in \{0,1\}^n$ one can output a circuit $D : \{0,1\}^\ell \to \{0,1\}^{3v+3}$ in time* $\mathrm{poly}(n, \log T)$ *mapping an index to a clause of a 3CNF ϕ in v-bit variables, for $v = \Theta(\ell)$, such that*

1. *ϕ is satisfiable iff there is $y \in \{0,1\}^T$ such that $M(x, y) = 1$, and*
2. *for any $r \leq n$ we can have $\ell = \max(\log T, n/r) + O(\log n) + O(\log \log T)$ and each output bit of D is a decision tree of depth $O(\log r)$.*

Note that by choosing $r := n/\log T$, for $T = 2^{\Omega(n)}$ we get that D is in NC^0 and ϕ has size $2^\ell = T \cdot \log^{O(1)} T$. We point out that the only place where locality $O(\log r)$ (as opposed to $O(1)$) is needed in D is to index bits of the string x.

The previous best result was D in NC^1 [BSGH+05]. Even in the simpler setting of $|\phi| = \mathrm{poly}(T)$ the previous best result was D in AC^0 [ASW09].

Our results simplify and tighten the aforementioned connection between satisfiability algorithms and lower bounds. In particular they simplify the lower bounds for ACC^0, by eliminating the workarounds mentioned above.

We also obtain tighter parameters. For example, a lower bound for depth d is implied by a satisfiability algorithm for depth $d + 2$, or even depth $d + 1$ in some cases [HP10]. By contrast previous workarounds required a satisfiability algorithm for depth $> 2d$. (In the original proof [Wil11b] this is due to the need of combining two guessed ACC^0 circuits, one computing the clauses and one the assignment; the loss in [SW12] appears even larger.). Moreover, the gates introduced by our reduction have constant fan-in. Another saving shows up in the size. Previous workarounds incurred a polynomial blow-up, while we are able to stay within linear size.

Just as PCP constructions have been optimized in order to obtain tight inapproximability results, it is conceivable that future lower bounds will benefit from more and more explicit reductions to 3SAT.

Next we formally state our improved connection and present the simplified proof. Recall that from non-trivial satisfiability algorithms for a circuit class one can get lower bounds on that class for functions in E^{NP}, NE, and NE $\cap$ coNE [Wil10, Wil11b, Wil13], and the smaller the class the weaker the lower bound. For simplicity we focus on the case of super-polynomial lower bounds on threshold circuits for E^{NP}. Recall that it is consistent with current knowledge that $\mathrm{EXP}^{\mathrm{NP}}$ has polynomial-size depth-2 circuits of unbounded-weight thresholds, which are a subclass of depth-3 circuits with polynomial-weight thresholds (aka majorities) [HMP+93, GHR92].

Theorem 2 (Tight connection between satisfiability and lower bounds)**.** *Consider unbounded fan-in circuits consisting of threshold gates (either bounded- or unbounded-weight).*

Suppose that for some constant d and for every c, given a circuit of depth $d+2$ and size n^c on n input bits one can decide its satisfiability in time $2^n/n^{\omega(1)}$.

Then E^{NP} does not have circuits of polynomial size and depth d.

Proof. Following [Wil10], suppose that E^{NP} has circuits of size n^c and depth d for some constants c and d. Let $L \in \mathrm{NTime}(2^n) \setminus \mathrm{NTime}(o(2^n))$ [Coo73, SFM78, Zák83]. Consider the E^{NP} algorithm that on input $x \in \{0,1\}^n$ and $i \le 2^n \mathrm{poly}(n)$ computes the 3CNF ϕ_x from Theorem 1, computes its first satisfying assignment if one exists, and outputs its ith bit. By assumption this algorithm can be computed by a circuit of depth d and size n^c. By hardwiring x we obtain that for every $x \in \{0,1\}^n$ there is a circuit C_x of the same depth and size that on input i computes that ith bit.

We contradict the assumption on L by showing how to decide it in $\mathrm{Ntime}(o(2^n))$. Consider the algorithm that on input $x \in \{0,1\}^n$ guesses the above circuit C_x. Then it connects three copies of C_x to the decision trees with depth $O(1)$ from Theorem 1 that on input $j \in \{0,1\}^{n+O(\log n)}$ compute the jth clause of ϕ_x in depth $O(\lg n)$, to obtain circuit C'_x. Since the paths in a decision tree are mutually exclusive, C'_x may be obtained simply by appending a $n^{O(1)}$-size layer of And gates to a layer of the gates of C_x, and increasing the fan-in of the latter, for a total depth of $d+1$. Then the algorithm constructs the circuit C''_x which in addition checks if the outputs of the 3 copies of C_x indeed satisfy the jth clause. The size of C''_x is $n^{O(c)}$ and a naive implementation yields depth $d+3$. Running the satisfiability algorithm on C''_x determines if ϕ_x is satisfiable and hence if $x \in L$ in time $2^{|j|}/|j|^{\omega(1)} = 2^n/n^{\omega(1)} = o(2^n)$.

We improve the depth of C''_x to $d+2$ as follows. The algorithm will guess instead of C_x a circuit D_x that given i and a bit b computes the ith bit mentioned above xor'ed with b. In an additional preliminary stage, the algorithm will check the consistency of D by running the satisfiability algorithm on (i) $D_x(i,0) \wedge D_x(i,1)$ and on (ii) $(\neg D_x(i,0)) \wedge \neg D_x(i,1))$, and reject if the output is ever 1. This circuit can be implemented in depth $d+1$. $\square$

Valiant's challenge [Val77] to exhibit an explicit function that cannot be computed by circuits of linear size and simultaneously logarithmic depth stands since 1977. In particular, it is still open whether E^{NP} has such circuits. By Theorem 1, similarly to the proof of Theorem 2, that can be ruled out by making progress on satisfiability algorithms for the same circuits.

Hansen and Podolskii [HP10] study depth-2 circuits with exact, unbounded-weight threshold gates, noting that lower bounds are not available. For this class our depth blow-up can be reduced to 1, by collapsing the output And gate with the outputs of the copies of D_x, see [HP10, Proposition 6].

Our results have a few other consequences. For example they imply that NC^0-explicit 3SAT is NEXP complete. Our techniques are also relevant to the notion of circuit uniformity. A standard notion of uniformity is log-space uniformity, requiring that the circuit is computable in logarithmic space or, equivalently, given an index to a gate in the circuit one can compute its type and its children in linear space. Equivalences with various other uniformity conditions are given by Ruzzo [Ruz81], see also [Vol99]. We consider another uniformity condition which is stronger than previously considered ones in some respects. Specifically, we describe the circuit by showing how to compute children by an NC^0 circuit, i.e. a function with constant locality.

Theorem 3 (L-uniform $\Leftrightarrow$ local-uniform)**.** *Let $f : \{0,1\}^* \to \{0,1\}$ be a function computable by a family of log-space uniform polynomial-size circuits. Then f is computable by a family of polynomial-size circuits $C = \{C_n : \{0,1\}^n \to \{0,1\}\}_n$ such that there is Turing machine that on input n (in binary) runs in time $O(\mathrm{poly}\log n)$ and outputs a map $D : \{0,1\}^{O(\log n)} \to \{0,1\}^{O(\log n)}$ such that*
(i) D has constant locality, i.e., every output bit of D depends on $O(1)$ input bits, and
(ii) on input a label g of a gate in C_n, D outputs the type of g and labels for each child.

1.2 Techniques

Background: Reducing non-deterministic time T to size-$\tilde{O}(T)$ 3SAT. Our starting point is the reduction of non-deterministic time-T computation to 3SAT instances of quasilinear size $T' = \tilde{O}(T)$. The classical proof of this result [HS66, Sch78, PF79, Coo88, GS89, Rob91, BSGH+05] hinges on the oblivious Turing machine simulation by Pippenger and Fischer [PF79]. However computing connections in the circuit induced by the oblivious Turing machine is a somewhat complicated recursive procedure, and we have not been able to use this construction for our results.

Instead, we use an alternative proof that replaces this simulation by coupling an argument by Gurevich and Shelah [GS89] with sorting networks. Recall the latter are sorting circuits, i.e., input-independent algorithms. This proof appears to be folklore. The first reference we are aware of is the survey by Van Melkebeek [vM06, §2.3.1], which uses Batcher's odd-even mergesort [Bat68]. This proof was rediscovered by a superset of the authors as a class project [VN12]. We now recall it.

Consider any general model of (non-deterministic) computation, such as RAM or random-access Turing machines. (One nice feature of this proof is that it directly handles models with random-access, aka direct-access, capabilities.) The proof reduces computation to the satisfiability of a circuit C. The latter is then reduced to 3SAT via the textbook reduction. Only the first reduction to circuit satisfiability is problematic and we will focus on that one here. Consider a non-deterministic time-T computation. The proof constructs a circuit of size $\tilde{O}(T)$ whose inputs are (non-deterministic guesses of) T configurations of the machine. Each configuration has size $O(\log T)$ and contains the state of the machine, all registers, and the content of the memory locations indexed by the registers. This computation is then verified in two steps. First, one verifies that every configuration C_i yields configuration C_{i+1} assuming that all bits read from memory are correct. This is a simple check of adjacent configurations. Then to verify correctness of read/write operations in memory, one sorts the configurations by memory indices, and within memory indices by timestamp. Now verification is again a simple check of adjacent configurations. The resulting circuit is outlined in Figure 2 (for a $2k$-tape random-access Turing machine). Using a sorting network of quasilinear size $\tilde{O}(T)$ results in a circuit of size $\tilde{O}(T)$.

Making low-space computation local. Our first new idea is a general technique that we call *spreading computation*. This shows that any circuit C whose connections can be computed in space linear in the description of a gate (i.e., space $\log|C|$) has an equivalent circuit C' of size $|C'| = \text{poly}|C|$ whose connections can be computed with constant locality. This technique is showcased in §2 in the simpler setting of Theorem 3.

The main idea in the proof is simply to let the gates of C' represent configurations of the low-space algorithm computing children in C. Then computing a child amounts to performing one step of the low-space algorithm, (each bit of) which can be done with constant locality in a standard Turing machine model. One complication with this approach is that the circuit C' has many invalid gates, i.e., gates that do not correspond to the computation of the low-space algorithm on a label of C. This is necessarily so, because constant locality is not powerful enough to even check the validity of a configuration. Conceivably, these gates could induce loops that do not correspond to computation, and make the final 3SAT instance always unsatisfiable. We avoid cycles by including a clock in the configurations, which allows us to ensure that each invalid gate leads to a sink.

We apply spreading computation to the various sub-circuits checking consistency of configurations, corresponding to the triangles in Figure 2. These sub-circuits operate on configurations of size $O(\log T)$ and have size $\text{poly}\log T$. Hence, we can tolerate the polynomial increase in their complexity given by the spreading computation technique.

There remain however tasks for which we cannot use spreading computation. One is the sorting sub-circuit. Since it has size $> T$ we cannot afford a polynomial increase. Another task is indexing adjacent configurations. We now discuss these two in turn.

Sorting network. We first mention a natural approach that gets us close but not quite to our main theorem. The approach is to define an appropriate labeling of the sorting network so that its connections can be computed very efficiently. We are able to define a labeling of bit-length $t + O(\log t) = \log \tilde{O}(T)$ for comparators in the odd-even mergesort network of size $\tilde{O}(2^t)$ (and depth t^2) that sorts $T = 2^t$ elements such that given a label one can compute the labels of its children by a decision tree of depth logarithmic in the length of the label, i.e. depth $\log\log \tilde{O}(T)$. With a similar labeling we can get linear size circuits. Or we can get constant locality at the price of making the 3SAT instance of size $T^{1+\epsilon}$. (Details omitted.)

To obtain constant locality we use a variant by Ben-Sasson, Chiesa, Genkin, and Tromer [BSCGT13]. They replace sorting networks with routing networks based on De Bruijn

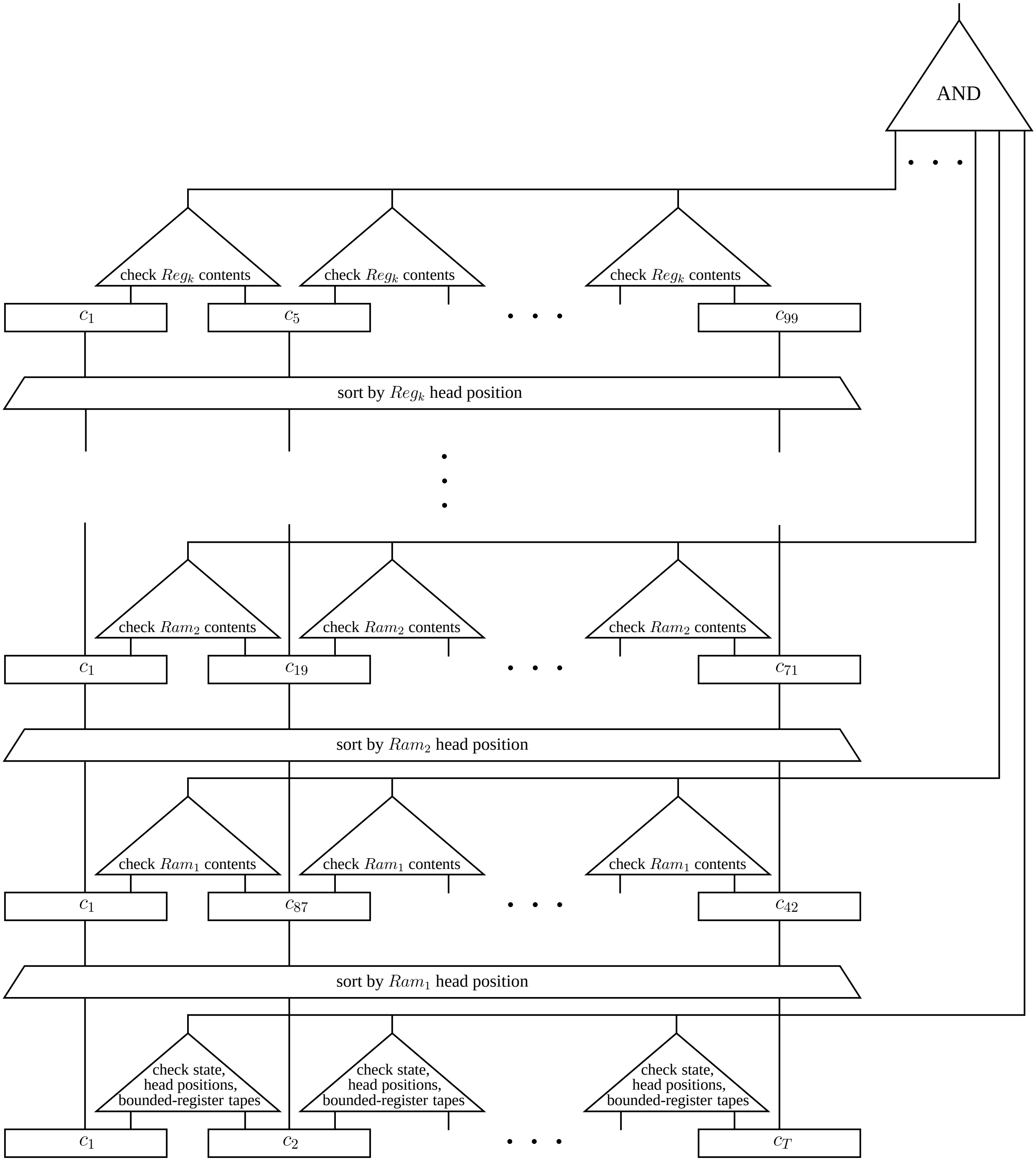

Figure 2: Each of the T configurations has size $O(\log T)$. The checking circuits have size poly $\log T$. The sorting circuits have size $\tilde{O}(T)$. k is a constant. Hence overall circuit has size $\tilde{O}(T)$.

graphs. They do so for their algebraic properties which are useful towards PCP constructions, whereas we exploit the small locality of these networks. Specifically, the connections of these networks involve computing bit-shift, bit-xor, and addition by 1. The first two operations can easily be computed with constant locality, but the latter cannot in the standard binary representation. However, this addition by 1 is only on $O(\log\log T)$ bits. Hence we can afford an alternative, redundant representation which gives us an equivalent network where all the operations can be computed with constant locality. This representation again introduces invalid labels; those are handled in a manner similar to our spreading computation technique.

Plus one. Regardless of whether we are using sorting or routing networks, another issue that comes up in all previous proofs is addition by 1 on strings of $> \log T$ bits. This is needed to index adjacent configurations C_i and C_{i+1} for the pairwise checks in Figure 2. As mentioned before, this operation cannot be performed with constant locality in the standard representation. Also, we cannot afford a redundant representation (since strings of length $c \log T$ would correspond to an overall circuit of size $> T^c$).

For context, we point out an alternative approach to compute addition by 1 with constant locality which however cannot be used because it requires an inefficient pre-processing. The approach is to use primitive polynomials over $\mathrm{GF}(2)^{\log T}$. These are polynomials modulo which x has order $2^{\log T} - 1$. Addition by 1 can then be replaced by multiplication by x, which can be shown to be local. This is similar to *linear feedback registers.* However, it is not known how to construct such polynomials efficiently w.r.t. their degrees, see [Sho92].

To solve this problem we use routing networks in a different way from previous works. Instead of letting the network output an array $C_1, C_2, \ldots$ representing the sorted configurations, we use the network to represent the "next configuration" map $C_i \to C_{i+1}$. Viewing the network as a matrix whose first column is the input and the last column is the output, we then perform the pairwise checks on every pair of input and output configurations that are in the same row. The bits of these configurations will be in the same positions in the final label, thus circumventing addition by one.

We mention that to simplify the proof in [Wil11b] it is sufficient to prove a weaker version of our Theorem 1 where C is in AC^0. For the latter, it essentially suffices to show that either the sorting network or the routing network's connections are in that class.

Organization. §2 showcases the spreading computation technique and contains the proof of Theorem 3. In §3 we present our results on routing networks. In §4 we discuss how to fetch the bits of the input x. §5 includes the proof of our main Theorem 1.

2 Spreading computation

In this section we prove Theorem 3.

Theorem 3 (L-uniform $\Leftrightarrow$ local-uniform)**.** *Let $f : \{0,1\}^* \to \{0,1\}$ be a function computable by a family of log-space uniform polynomial-size circuits. Then f is computable by a family of polynomial-size circuits $C = \{C_n : \{0,1\}^n \to \{0,1\}\}_n$ such that there is Turing machine that on input n (in binary) runs in time $O(\text{poly}\log n)$ and outputs a map $D : \{0,1\}^{O(\log n)} \to \{0,1\}^{O(\log n)}$ such that*
(i) D has constant locality, i.e., every output bit of D depends on $O(1)$ input bits, and
(ii) on input a label g of a gate in C_n, D outputs the type of g and labels for each child.

We will use the following formalization of log-space uniformity: a family of polynomial-size circuits $C' = \{C'_n : \{0,1\}^n \to \{0,1\}\}_n$ is log-space uniform if there exists a Turing machine M that, on input $g \in \{0,1\}^{\log |C'_n|}$ labeling a gate in C'_n, and n written in binary, uses space $O(\log n)$ and outputs the types and labels of each of g's children. (Note that M outputs the types of g's children rather than g's type; the reason for this will be clear from the construction below.)

Proof. Let C' be a log-space uniform family of polynomial-size circuits and M a log-space machine computing connections in C'. We make the following simplifying assumptions without loss of generality.

- Each gate in C' has one of the following five types: And (fan-in-2), Not (fan-in-1), Input (fan-in-0), Constant-0 (fan-in-0), Constant-1 (fan-in-0).
- For all n, $|C'_n|$ is a power of 2. In particular, each $(\log |C'_n|)$-bit string is a valid label of a gate in C'_n.
- M's input is a label $g \in \{0,1\}^{\log |C'_n|}$, a child-selection-bit $c \in \{0,1\}$ that specifies which of g's ≤ 2 children it should output, and n in binary. M terminates with $\log |C'_n|$ bits of its tape containing the child's label, and 3 bits containing the child's type.

The local-uniform family C will additionally have fan-in-1 Copy gates that compute the identity function. Gates in C are labeled by configurations of M, and we now specify these. Let $q, k, k' = O(1)$ be such that M has $2^q - 1$ states, and on input $(g, c, n) \in \{0,1\}^{O(\log |C'_n|)}$ it uses space $\le k \log n$ and runs in time $\le n^{k'}$. A configuration of M is a bit-string of length $((q+2) \cdot k + 2k') \cdot \log n$, and contains two items: the *tape* and the *timestep*.

The tape is specified with $(q+2) \cdot k \cdot \log n$ bits. Each group of $q+2$ bits specifies a single cell of M's tape as follows. The first two bits specify the value of the cell, which is either 0, 1, or blank. The remaining q bits are all zero if M's head is not on this cell, and otherwise they contain the current state of M.

The timestep is specified with $2k' \cdot \log n$ bits. In order to allow it to be incremented by a local function, we use the following representation which explicitly specifies the carry bits arising from addition. View the timestep as a sequence of pairs

$$((c_{k'\log n}, b_{k'\log n}), (c_{k'\log n - 1}, b_{k'\log n - 1}), \ldots, (c_1, b_1)) \in \{0,1\}^{2k'\log n}.$$

Then the timestep is initialized with $c_i = b_i = 0$ for all i, and to increment by 1 we simultaneously set $c_1 \leftarrow b_1$, $b_1 \leftarrow b_1 \oplus 1$, and $c_i \leftarrow b_i \wedge c_{i-1}$ and $b_i \leftarrow b_i \oplus c_{i-1}$ for all $i > 1$.

It is not difficult to see that there is a local map $\mathsf{Upd} : \{0,1\}^{O(\log n)} \to \{0,1\}^{O(\log n)}$ that, on input a configuration of M, outputs the configuration that follows in a single step. Namely Upd increments the timestep using the method described above, and updates each cell of the tape by looking at the $O(1)$ bits representing that cell and the two adjacent cells.

We say that a configuration is *final* iff the most-significant bit of the timestep is 1. This convention allows a local function to check if a configuration is final. Using the above method for incrementing the timestep, a final configuration is reached after $n^{k'} + k' \log n - 1$ steps. We say that a configuration is *valid* if either (a) it is the initial configuration of M on some input $(g, c) \in \{0,1\}^{\log |C'_n|+1}$ labeling a gate in C'_n and specifying one of its children, or (b) it is reachable from such a configuration by repeatedly applying Upd. (Note that Upd must be defined on every bit-string of the appropriate length. This includes strings that are not valid configurations, and on these it can be defined arbitrarily.)

We now describe the circuit family $C = \{C_n\}_n$ and the local map D that computes connections in these circuits, where D depends on n.

C_n has size n^u for $u := (q+2) \cdot k + 2k' = O(1)$, and each gate is labeled by an $(u \log n)$-bit string which is parsed as a configuration of M. C_n is constructed from C'_n by introducing a chain of Copy gates between each pair of connected gates $(g_{\mathsf{parent}}, g_{\mathsf{child}})$ in C'_n, where the gates in this chain are labeled by configurations that encode the computation of M on input g_{parent} and with output g_{child}.

Let $g \in \{0,1\}^{u \log n}$ be a configuration of M labeling a gate in C_n. Our convention is that if g is a final configuration then the type of g is what is specified by three bits at fixed locations on M's tape, and if g is not a final configuration then the type is Copy. (Recall that when M terminates, the type of its output is indeed written in three bits at fixed locations.) In particular, the type of a gate can be computed from its label g by a local function.

D computes the children of its input g as follows. If g is not a final configuration, then D outputs the single child whose configuration follows in one step from g using the map Upd described above. If g is a final configuration, D first determines its type and then proceeds as follows. If the type is And, then D outputs two children by erasing all but the $\log |C'_n|$ bits of M's tape corresponding to a label of a gate in C'_n, writing n, setting the timestep to 0, putting M in its initial state with the head on the leftmost cell, and finally setting one child to have $c = 0$ and one child to have $c = 1$. (Recall that c is the child-selection-bit for M.) If the type is Not, then D acts similarly but only outputs the one with $c = 0$. For any other type, g has fan-in 0 and thus D outputs no children.

Naturally, the output gate of C_n is the one labeled by the configuration consisting of the first timestep whose MSB is 1 and the tape of M containing $(g_{\mathsf{out}}, t_{\mathsf{out}}, n)$ where g_{out} is the unique label of C''s output gate and t_{out} is its type. (The remainder of this configuration can be set arbitrarily.) It is clear that starting from this gate and recursively computing all children down to the fan-in-0 gates of C_n gives a circuit that computes the same function as C'_n. Call the tree computed in this way the *valid tree*.

We observe that C_n also contains gates outside of the valid tree, namely all gates whose labels do not correspond to a valid configuration. To conclude the proof, we show that the topology of these extra gates does not contain any cycles, and thus C_n is a valid circuit.

By avoiding cycles, we ensure that the circuit can be converted to a constraint-satisfaction problem (i.e. 3SAT); the existence of a cycle with an odd number of Not gates would cause the formula to be always unsatisfiable.

Consider a label g of a gate in C_n containing a configuration of M. If g is the label of a gate in the valid tree, then it is clearly not part of a cycle. If g is any other label, we consider two cases: either g is a final configuration or it is not. If g is not a final configuration, then its descendants eventually lead to a final configuration g'. (This follows because of the inclusion of the timestep in each configuration, and the fact that starting from any setting of the (c_i, b_i) bits and repeatedly applying the increment procedure will eventually yield a timestep with MSB = 1.) Notice that the tape in g' contains $\log |C'_n|$ bits corresponding to a valid label of a gate in C'_n. (This is because of our convention that any bit-string of that length is a valid label. An alternative solution intuitively connects the gates with MSB = 1 to a sink, but has other complications.) Therefore the children of g' are in the valid tree, and so g' (and likewise g) is not part of a cycle. Similarly, if g is a final configuration then its children are in the valid tree and so it is not part of a cycle. $\square$

3 Routing networks

In this section we show how to non-deterministically implement the sorting subcircuits. We do this in a way so that for every input sequence of configurations, at least one non-deterministic choice results in a correctly sorted output sequence. Further, each possible output sequence either is a permutation of the input or contains at least one "dummy configuration" (wlog the all-zero string). Importantly, the latter case can be detected by the configuration-checking subcircuits.

Theorem 4. *Fix $T = T(n) \geq n$. Then for all $n > 0$, there is a circuit*

$$S : \left(\{0,1\}^{O(\log T)}\right)^T \times \{0,1\}^{O(T \log T)} \to \left(\{0,1\}^{O(\log T)}\right)^T$$

of size $T' := T \cdot \log^{O(1)} T$ and a labelling of the gates in S by strings in $\{0,1\}^{\log T'}$ such that the following holds.

1. *There is a local map $D : \{0,1\}^{\log T'} \times \{0,1\} \to \{0,1\}^{\log T'}$ such that for every label g of a gate in S, $D(g, b)$ outputs the label of one of g's ≤ 2 children (according to b). Further, the type of each gate can be computed from its label in NC^0. The latter NC^0 circuit is itself computable in time* $\operatorname{poly}\log T$.

2. *Given a $(\log T + O(\log\log T))$-bit index into S's output, the label of the corresponding output gate can be computed in* NC^0. *Further, given any input gate label, the corresponding $(\log T + O(\log\log T))$-bit index into the input can be computed in* NC^0. *These two NC^0 circuits are computable in time* $\operatorname{poly}\log T$.

3. *For every $C = (C_1, ..., C_T)$ and every permutation $\pi : [T] \to [T]$, there exists z such that $S(C, z) = (C_{\pi(1)}, \ldots, C_{\pi(T)})$.*

4. *For every $C = (C_1, ..., C_T)$ and every z, if $(C'_1, \ldots, C'_T) := S(C, z)$ is not a permutation of the input then for some i, C'_i is the all-zero string.*

We construct this circuit S using a *routing network*.

Definition 5. Let G be a directed layered graph with ℓ columns, m rows, and edges only between subsequent columns such that each node in the first (resp. last) $\ell - 1$ columns has exactly two outgoing (resp. incoming) edges.

G is a *routing network* if for every permutation $\pi : [m] \to [m]$, there is a set of m node-disjoint paths that link the i-th node in the first column to the $\pi(i)$-th node in the last column, for all i.

Our circuit S will be a routing network in which each node is a 2×2 switch that either direct- or cross-connects (i.e. flips) its input pair of configurations to its output pair, depending on the value of an associated control bit. This network is used to non-deterministically sort the input sequence by guessing the set of control bits. We use routing networks constructed from De Bruijn graphs as given in [BSCGT13].

Definition 6. An *n-dimensional De Bruijn graph DB_n* is a directed layered graph with $n + 1$ columns and 2^n rows. Each node is labeled by (w, i) where $w \in \{0, 1\}^n$ specifies the row and $0 \leq i \leq n$ specifies the column. For $i < n$, each node (w, i) has outgoing edges to $(\mathsf{sr}(w), i + 1)$ and $(\mathsf{sr}(w) \oplus 10 \cdots 0, i + 1)$, where sr denotes cyclic right shift.

A *k-tandem n-dimensional De Bruijn graph DB_n^k* is a sequence of k n-dimensional De Bruijn graphs connected in tandem.

Theorem 7 ([BSCGT13])**.** *For every n, DB_n^4 is a routing network.*

To use this in constructing the sorting circuit S, we must show how, given the label (w, i) of any node in DB_n^4, to compute in NC^0 the labels of its two predecessors.

Computing the row portion of each label (corresponding to w) is trivially an NC^0 operation, as w is mapped to $\mathsf{sl}(w)$ and $\mathsf{sl}(w \oplus 10 \cdots 0)$ where sl denotes cyclic left shift.

For the column portion (corresponding to i), we use the encoding of integers from Theorem 3 that explicitly specifies the carry bits arising from addition. Namely, we use a $(2\log(4n))$-bit counter as described there, and number the columns in reverse order so that the last column is labeled by the initial value of the counter and the first column is labeled by the maximum value. This actually results in more columns than needed, specifically $4n + \log(4n)$, due to the convention that the counter reaches its maximum when the MSB becomes 1. (We use this convention here to determine when we are at the input level of the circuit.) However, note that adding more columns to DB_n^4 does not affect its rearrangeability since whatever permutation is induced by the additional columns can be accounted for by the rearrangeability of the first $4n + 1$ columns.

The next proof will introduce some dummy configurations whose need we explain now. With any routing network, one can talk about either edge-disjoint routing or node-disjoint routing. Paraphrasing [BSCGT13, first par after Def A.6], a routing network with m rows can be used to route $2m$ configurations using edge-disjoint paths (where each node receives and sends two configs), or m configurations using node-disjoint paths (where each node receives and sends one configuration). In the former every edge carries a configuration, while in the latter only half the edges between each layer carry configurations (which half depends on the permutation being routed). However, when implementing either type of routing with a boolean circuit, all edges must of course always be present and carry some data, because they correspond to wires. Thus for node-disjoint routing, half of the edges between each layer carry "dummy configurations" in our construction, and it is possible even for the dummy configurations to appear at the output of the network for certain (bad) settings of the switches. This whole issue would be avoided with edge-disjoint routing (which is arguably more natural when implementing routing networks with circuits), but we prefer to rely on existing proofs as much as possible.

Proof of Theorem 4. The circuit S is a De Bruijn graph with T rows and $4\log T+\log(4\log T)$ columns as described above. It routes the T configurations specified by its first input according to the set of paths specified by its second input. Each node not in the first or last column is a 2×2 switch on $O(\log T)$-bit configurations with an associated control bit from S's second input specifying whether to swap the configurations. Each node in the last column has a control bit that selects which of its two inputs to output. Nodes in the first column map one input to two outputs; these have no control bits, and output their input along with the all-zero string.

We label each non-input gate in S by $(t = 00, w, i, s, d)$ where $t = 00$ specifies "non-input", $(w, i) \in \{0,1\}^{\log T} \times \{0,1\}^{O(\log\log T)}$ specifies a switch (i.e. a node in the De Bruijn graph), and $(s, d) \in \{0,1\}^{O(\log\log T)} \times \{0,1\}^{O(1)}$ specifies a gate within this switch. For the latter, we view a switch on $O(\log T)$-bit configurations as $O(\log T)$ switches on individual bits; then s designates an $O(1)$-sized bit switch, and d designates a gate within it.

We label each gate in S's first input by $(t = 01, w, s)$ where $t = 01$ specifies "first input", $w \in \{0,1\}^{\log T}$ specifies one of the T configurations, and $s \in \{0,1\}^{O(\log\log T)}$ specifies a bit within the configuration.

We label each gate in S's second input by $(t = 10, w, i)$ where $t = 10$ specifies "second input" and $(w, i) \in \{0,1\}^{\log T} \times \{0,1\}^{O(\log\log T)}$ specifies a switch.

We take any gate with $t = 11$ to be a Constant-0 gate, one of which is used to output the all-zero string in the first column.

Naturally the labels of S's output gates vary over w and s and have $t = 00$, $i = 0\cdots 0$, and $d =$ the output gate of a bit switch; these and the input gate labels above give Property 2. Theorem 7 guarantees that Property 3 holds for some setting of the switches, and it is straightforward to verify that Property 4 holds for any setting of the switches. We now show Property 1, namely how to compute connections in S with a local map D.

Suppose $g = (t = 00, w, i, s, d)$ is the label of a non-input gate, and let $b \in \{0,1\}$ select one of its children. There are four possible cases: (1) the child is in the same 2×2 bit switch, (2) the child is an output gate of a preceding 2×2 bit switch, (3) the child is a bit of a configuration from S's first input or the all-zero string, or (4) the child is a control bit from S's second input. Since each bit switch has a fixed constant-size structure, the case can be determined by reading the $O(1)$ bits corresponding to d and the MSB of i which specifies whether g is in the first column of the De Bruijn graph.

For case (1), D updates d to specify the relevant gate within the bit switch. For case (2), D updates w and i via the procedures described above, and updates d to specify the output gate of the new bit switch. For cases (3) and (4), D updates t and either copies the relevant portions (w, s or w, i) from the rest of g if $t \neq 11$, or sets the rest of g to all zeros if $t = 11$.

Finally we note that as in Theorem 3, there are strings that do not encode valid labels in the manner described above, and that these do not induce any cycles in the circuit S due to the way the field i is used. □

4 Fetching bits

In this section, we construct a local-uniform circuit that will be used to fetch the bits of the fixed string $x \in \{0,1\}^n$ in our final construction. Moreover, we demonstrate a trade-off between the length of the labels and the locality of the map between them.

Theorem 8. *For all $x \in \{0,1\}^n$ and for all $r \in [n]$, there is a circuit C of size 2^ℓ where $\ell = n/r + O(\log n)$, a labeling of the gates in C by strings in $\{0,1\}^\ell$, and a map $D : \{0,1\}^\ell \to \{0,1\}^\ell$ each bit of which is computable by a decision tree of depth $O(\log r)$ with the following properties:*

1. *All gates in C are either fan-in-0 Constant-0 or Constant-1 gates, or fan-in-1 Copy gates. In particular, C has no input gates.*

2. *There are n output gates $\mathsf{out}_1, \ldots, \mathsf{out}_n$, a Constant-0 gate g_0, and a Constant-1 gate g_1 such that, for all $i \leq n$, repeatedly applying D to the label of out_i eventually yields the label of g_{x_i}.*

3. *$\forall i \leq n$: the label of out_i can be computed in NC^0 from the binary representation of i.*

4. *Given x and r the decision trees computing D, and the NC^0 circuit in the previous item can be computed in time $\mathrm{poly}(n)$.*

Proof. We first explain the high-level idea of the construction. Assume $r = 1$, we later extend this to the general case. For each i, the chain of labels induced by D starting from the label of out_i will encode the following process. Initialize an n-bit string $s := 10\cdots0$ of Hamming weight 1. Then shift s to the right $i-1$ times, bit-wise AND it with x, and finally shift it to the left $i-1$ times. Clearly, the leftmost bit of s at the end of this process will equal x_i. The main technical difficulty is encoding counting to i for arbitrary i while allowing connections in C to be computed locally. We achieve this using similar techniques as in the proof of Theorem 3 in Section 2, namely by performing the counting with a machine M whose configurations we store in the labels of C. We now give the details.

The label of each gate in C is parsed as a tuple (t, s, d, i, c) of length $\ell = n + O(\log n)$ as follows: t is a 2-bit string specifying the type of the gate, s is the n-bit string described above, d is a 1-bit flag specifying the direction s is currently being shifted (left or right), i is the

$(\log n)$-bit binary representation of an index into x, and c is the $O(\log n)$-bit configuration of a machine M that operates as follows. M has $\log n$ tape cells initialized to some binary number. It decrements the number on its tape, moves its head to the left-most cell and enters a special state q^*, and then repeats this process, terminating when its tape contains the binary representation of 1. We encode M's $O(\log n)$-bit configurations as in Theorem 3, in particular using the same timestep format so that checking if M has terminated can be done by reading a single bit.

The label of out_i has the following natural form. The type t is Copy, s is initialized to $10\cdots0$, the flag d encodes "moving right", i is the correct binary representation, and c is the initial configuration of M on input i. Note that this can be computed from i in NC^0.

The local map D simply advances the configuration c, and shifts s in the direction specified by d iff it sees the state q^* in M's left-most cell. If c is a final configuration and d specifies "moving right", then D bit-wise ANDs x to s, sets d to "moving left", and returns M to its initial configuration on input i. If c is a final configuration and d specifies "moving left", then D outputs the unique label of the constant gate g_b where b is the left-most bit of s. (Without loss of generality, we can take this to be the label with the correct type field and all other bits set to 0.)

The correctness of this construction is immediate. Furthermore, the strings that encode invalid labels do not induce cycles in C for similar reasons as those given at the end of Theorem 3. (In fact, the presence of cycles in this component would not affect the satisfiability of our final 3SAT instance, since the only gates with non-zero fan-in have type Copy.)

We now generalize the proof to any value of r. The goal is to establish a trade-off between the label length ℓ and the locality of the map D such that at one extreme we have $\ell = n + O(\log n)$ and D of constant locality and the the other we have $\ell = O(\log n)$ and D computable by decision trees of depth $O(\log n)$.

The construction is the same as before but this time the label of a gate in C is parsed as a tuple (t, p, k, d, i, c) of length $\ell = n/r + \log r + O(\log n) = n/r + O(\log n)$, where t, d, i, and c are as before and $p \in \{0,1\}^{n/r}$ and $k \in \{0,1\}^{\log r}$ together represent a binary string of length n and Hamming weight 1. More precisely, consider a binary string $s \in \{0,1\}^n$ of Hamming weight 1 partitioned into r segments each of n/r bits. Now, the position of the bit set to 1 can be determined by a segment number $k \in \{0,1\}^{\log r}$ and a bit string $p \in \{0,1\}^{n/r}$ of Hamming weight 1 within the segment.

The map D now *cyclically* shifts the string p in the direction indicated by d, updating k as needed. For the rest, the behavior of D remains unchanged. In particular, If c is a final configuration and d specifies "moving right", then D bit-wise ANDs the relevant n/r-bit segment of x to p and so on. To perform one such step, D needs to read the entire k in addition to a constant number of other bits, so it can be computed by decision trees of depth $O(\log r)$. $\square$

5 Putting it together

We now put these pieces together to prove Theorem 1. First we modify previous proofs to obtain the following normal form for non-deterministic computation that is convenient for our purposes, cf. §1.2.

Theorem 9. *Let M be an algorithm running in time $T = T(n) \geq n$ on inputs of the form (x, y) where $|x| = n$. Then there is a function $T' = T \log^{O(1)} T$, a constant $k = O(1)$, and k logspace-uniform circuit families $C_1, \ldots, C_k$ each of size $\log^{O(1)} T$ with oracle access to x, such that the following holds:*

For every $x \in \{0,1\}^n$, there exists y such that $M(x, y)$ accepts in $\leq T$ steps iff there exists a tuple $(z_1, \ldots, z_{T'}) \in \left(\{0,1\}^{O(\log T)}\right)^{T'}$, and k permutations $\pi_1, \ldots, \pi_k : [T'] \to [T']$ such that for all $j \leq k$ and $i \leq T'$, $C_j\left(z_i, z_{\pi_j(i)}\right)$ outputs 1.

We note that "oracle access to x" means that the circuits have special gates with $\log n$ input wires that output x_i on input $i \leq n$ represented in binary. Alternatively the circuits C_i do not have oracle access to x but instead there is a separate constraint that, say, the first bit of z_i equals x_i for every $i \leq n$.

Proof sketch. Model M as a random-access Turing machine running in time T' and using indices of $O(\log T') = O(\log T)$ bits. All standard models of computation can be simulated by such machines with only a polylogarithmic factor T'/T blow-up in time. Each z_i is an $O(\log T)$-bit configuration of M on some input (x, y). This configuration contains the timestamp $i \leq T'$, the current state of M, the indices, and the contents of the indexed memory locations; see [VN12] for details.

The circuits and permutations are used to check that $(z_1, \ldots, z_{T'})$ encodes a valid, accepting computation of $M(x, y)$. This is done in $k+1$ phases where $k = O(1)$ is the number of tapes. First, we use C_1 to check that each configuration z_i yields z_{i+1} assuming that all bits read from memory are correct, and to check that configuration $z_{T'}$ is accepting. (For this we use the permutation $\pi_1(i) := i + 1 \mod T'$.) This check verifies that the state, timestamp, and indices are updated correctly. To facilitate the subsequent checks, we assume without loss of generality that M's first n steps are a pass over its input x. Therefore, C_1 also checks (using oracle access to x) that if the timestamp i is $\leq n$ then the first index has value i and the bit read from memory is equal to x_i.

For $j > 1$, we use C_j to verify the correctness of the read/write operations in the $(j-1)$-th tape. To do this, we use the permutation π_j such that for each i, z_i immediately precedes $z_{\pi_j(i)}$ in the sequence of configurations that are sorted first by the $(j-1)$-th index and then by timestamp. Then, C_j checks that its two configurations are correctly sorted, and that if index $j-1$ has the same value in both then the bit read from memory in the second is consistent with the first. It also checks that the value of any location that is read for the first time is blank, except for the portion on the first tape that corresponds to the input (x, y). (Note that C_1 already verified that the first time M reads a memory index $i \leq n$, it contains x_i. No checks is performed on the y part, corresponding to this string being existentially quantified.)

We stipulate that each C_j above outputs 0 if either of its inputs is the all-zero string, which happens if the sorting circuit does not produce a permutation of the configurations (cf. Theorem 4, part 4). Finally, we observe that all checks can be implemented by a log-space uniform family of polynomial-size circuits with oracle access to x. $\square$

We now prove our main theorem, restated for convenience. The high-level idea is to use §2-4 to transform the circuits from Theorem 9 into circuits whose connections are computable by small-depth decision trees, and to then apply the textbook reduction from Circuit-SAT to 3SAT.

Theorem 1 (Local reductions). *Let M be an algorithm running in time $T = T(n) \geq n$ on inputs of the form (x, y) where $|x| = n$. Given $x \in \{0,1\}^n$ one can output a circuit $D : \{0,1\}^\ell \to \{0,1\}^{3v+3}$ in time $\mathrm{poly}(n, \log T)$ mapping an index to a clause of a 3CNF ϕ in v-bit variables, for $v = \Theta(\ell)$, such that*

1. *ϕ is satisfiable iff there is $y \in \{0,1\}^T$ such that $M(x, y) = 1$, and*

2. *for any $r \leq n$ we can have $\ell = \max(\log T, n/r) + O(\log n) + O(\log\log T)$ and each output bit of D is a decision tree of depth $O(\log r)$.*

Proof. We parse D's input as a tuple (g, r, s), where g is the label of a gate in some component from Theorem 9, as explained next, r is a 2-bit clause index, and s is a 1-bit control string. We specifically parse g as a pair (Region, Label) as follows. Region (hereafter, R) is an $O(1)$-bit field specifying that Label is the label of either

(a) a gate in a circuit that implements the ith instance of some C_j,

(b) a gate in a circuit that provides oracle access to x,

(c) a gate in a circuit that implements some π_j via a routing network, or

(d) a gate providing a bit of some configuration z_i.

Label (hereafter, L) is a $(\max(\log T, n/r) + O(\log n) + O(\log\log T))$-bit field whose interpretation varies based on R. For (a), we take $L = (i, j, \ell)$ where $i \leq T$ and $j \leq k$ specify $C_j(z_i, z_{\pi_j(i)})$ and $\ell \in \{0,1\}^{O(\log\log T)}$ specifies a gate within it, where we use Theorem 3 and take C_j to be a circuit whose connections are computable in NC^0. For (b), we take L to be a $(n/r + O(\log n))$-bit label of the circuit from Theorem 8. For (c), we take $L = (j, \ell)$ where $j \leq k$ specifies π_j and $\ell \in \{0,1\}^{\log T + O(\log\log T)}$ specifies a gate in the circuit from Theorem 4 implementing π_j. For (d), L is simply the $(\log T + O(\log\log T))$-bit index of the bit.

We now describe D's computation. First note that from Theorems 3, 4, and 8, the type of g can be computed from L in NC^0; call this value Type $\in$ {And, Not, Copy, Input, x-Oracle, Constant-0, Constant-1}.

Computing g's children. D first computes the labels of the ≤ 2 children of the gate $g = (R, L)$ as follows.

If R specifies that $L = (i, j, \ell)$ is the label of a gate in $C_j(z_i, z_{\pi_j(i)})$, D computes ℓ's child(ren) using the NC^0 circuit given by Theorem 3. The only cases not handled by this are when Type $\in$ {x-Oracle, Input}. When Type $=$ x-Oracle, the child is the i'th output gate of the bit-fetching circuit, where i' is the lower $\log n$ bits of i; by part 3 of Theorem 8, the label of this gate can be computed in NC^0. When Type $=$ Input, the child is either the mth bit of z_i or the mth bit of π_j's ith output, for some $m \leq O(\log T)$. We assume without loss of generality that m is contained in binary in a fixed position in L, and that which of the two inputs is selected can be determined by reading a single bit of L. Then, the label of the bit of z_i can be computed in NC^0 by concatenating i and m, and the label of the mth bit of π_j's ith output can be computed by part 2 of Theorem 4.

If R specifies that L is a label in the bit-fetching circuit from Theorem 8, D computes its child using the $O(\log r)$-depth decision trees given by that theorem.

If R specifies that $L = (j, \ell)$ is the label of a sorting circuit from Theorem 4, D computes ℓ's child(ren) using the NC^0 circuit given by that theorem. The only case not handled by this is when ℓ labels a gate in the first input to the sorting circuit, but in this case the child is a bit of some z_i where i can be computed in NC^0 by part 2 of Theorem 4.

If Type $=$ Input and (R, L) is not one of the cases mentioned above or Type $\in$ {Constant-0, Constant-1}, D computes no children.

Outputting the clause. When the control string $s = 0$, D outputs the clause specified by g and r in the classical reduction to 3SAT, which we review now. (Recall that r is a 2-bit clause index.) The 3SAT formula ϕ contains a variable for each gate g, including each input gate, and the clauses are constructed as follows.

If Type $=$ And, we denote g's children by g_a and g_b. Then depending on the value of r, D outputs one of the four clauses in the formula

$$(g_a \vee g_b \vee \overline{g}) \wedge (g_a \vee \overline{g_b} \vee \overline{g}) \wedge (\overline{g_a} \vee g_b \vee \overline{g}) \wedge (\overline{g_a} \vee \overline{g_b} \vee g).$$

These ensure that in any satisfying assignment, $g = g_a \wedge g_b$.

If Type $=$ Not, we denote g's child by g_a. Then depending on the value of r, D outputs one of the two clauses in the formula

$$(g \vee g_a \vee g_a) \wedge (\overline{g} \vee \overline{g_a} \vee \overline{g_a}).$$

These ensure that in any satisfying assignment, $g = \overline{g_a}$.

If Type $\in$ {x-Oracle, Copy} or Type $=$ Input and D computed g's child g_a, then depending on the value of r, D outputs one of the two clauses in the formula

$$(\overline{g} \vee g_a \vee g_a) \wedge (g \vee \overline{g_a} \vee \overline{g_a}).$$

These ensure that in any satisfying assignment, $g = g_a$.

If Type $=$ Constant-0, D outputs the clause $(\overline{g} \vee \overline{g} \vee \overline{g})$ which ensures that in any satisfying assignment g is false (i.e. that each Constant-0 gate outputs 0). If Type $=$ Constant-1, D

outputs the clause $(g \vee g \vee g)$ which ensures that in any satisfying assignment g is true (i.e. that each Constant-1 gate outputs 1).

If Type = Input, and D did not compute a child of g, D outputs a dummy clause $(g_{\mathsf{dummy}} \vee g_{\mathsf{dummy}} \vee g_{\mathsf{dummy}})$ where g_{dummy} is a string that is distinct from all other labels g.

When the control string $s = 1$, D outputs clauses encoding the restriction that each $C_j(z_i, z_{\pi_j(i)})$ outputs 1. Namely, D parses $L = (i, j, \ell)$ as above, and outputs $(g_{i,j} \vee g_{i,j} \vee g_{i,j})$, where $g_{i,j} := (i, j, \ell^*)$ and ℓ^* is the label of C_j's output gate, which depends only on j and $\log T$ and thus can be hardwired into D. $\square$

Acknowledgments. We are very grateful to Eli Ben-Sasson for a discussion on routing networks which led us to improving our main result, cf. §1.2.

References

[AAI+01] Manindra Agrawal, Eric Allender, Russell Impagliazzo, Toniann Pitassi, and Steven Rudich. Reducing the complexity of reductions. *Computational Complexity*, 10(2):117–138, 2001.

[AK10] Eric Allender and Michal Koucký. Amplifying lower bounds by means of self-reducibility. *J. of the ACM*, 57(3), 2010.

[ASW09] Sanjeev Arora, David Steurer, and Avi Wigderson. Towards a study of low-complexity graphs. In *Coll. on Automata, Languages and Programming (ICALP)*, pages 119–131, 2009.

[Bat68] Kenneth E. Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Computing Conference*, volume 32, pages 307–314, 1968.

[BSCGT12] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, and Eran Tromer. On the concrete-efficiency threshold of probabilistically-checkable proofs. *Electronic Colloquium on Computational Complexity (ECCC)*, 19:45, 2012.

[BSCGT13] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, and Eran Tromer. Fast reductions from RAMs to delegatable succinct constraint satisfaction problems. In *ACM Innovations in Theoretical Computer Science conf. (ITCS)*, pages 401–414, 2013.

[BSGH+05] Eli Ben-Sasson, Oded Goldreich, Prahladh Harsha, Madhu Sudan, and Salil P. Vadhan. Short pcps verifiable in polylogarithmic time. In *IEEE Conf. on Computational Complexity (CCC)*, pages 120–134, 2005.

[Coo73] Stephen A. Cook. A hierarchy for nondeterministic time complexity. *J. of Computer and System Sciences*, 7(4):343–353, 1973.

[Coo88] Stephen A. Cook. Short propositional formulas represent nondeterministic computations. *Information Processing Letters*, 26(5):269–270, 1988.

[FLvMV05] Lance Fortnow, Richard Lipton, Dieter van Melkebeek, and Anastasios Viglas. Time-space lower bounds for satisfiability. *J. of the ACM*, 52(6):835–865, 2005.

[GHR92] Mikael Goldmann, Johan Håstad, and Alexander A. Razborov. Majority gates vs. general weighted threshold gates. *Computational Complexity*, 2:277–300, 1992.

[GS89] Yuri Gurevich and Saharon Shelah. Nearly linear time. In *Logic at Botik, Symposium on Logical Foundations of Computer Science*, pages 108–118, 1989.

[HMP+93] András Hajnal, Wolfgang Maass, Pavel Pudlák, Márió Szegedy, and György Turán. Threshold circuits of bounded depth. *J. of Computer and System Sciences*, 46(2):129–154, 1993.

[HP10] Kristoffer Arnsfelt Hansen and Vladimir V. Podolskii. Exact threshold circuits. In *IEEE Conf. on Computational Complexity (CCC)*, pages 270–279, 2010.

[HS66] Fred Hennie and Richard Stearns. Two-tape simulation of multitape turing machines. *J. of the ACM*, 13:533–546, October 1966.

[Mie09] Thilo Mie. Short pcpps verifiable in polylogarithmic time with $o(1)$ queries. *Ann. Math. Artif. Intell.*, 56(3-4):313–338, 2009.

[PF79] Nicholas Pippenger and Michael J. Fischer. Relations among complexity measures. *J. of the ACM*, 26(2):361–381, 1979.

[Rob91] J. M. Robson. An O(T log T) reduction from RAM computations to satisfiability. *Theoretical Computer Science*, 82(1):141–149, 1991.

[Ruz81] Walter L. Ruzzo. On uniform circuit complexity. *J. of Computer and System Sciences*, 22(3):365–383, 1981.

[Sch78] Claus-Peter Schnorr. Satisfiability is quasilinear complete in NQL. *J. of the ACM*, 25(1):136–145, 1978.

[SFM78] Joel I. Seiferas, Michael J. Fischer, and Albert R. Meyer. Separating nondeterministic time complexity classes. *J. of the ACM*, 25(1):146–167, 1978.

[Sho92] Victor Shoup. Searching for primitive roots in finite fields. *Math. Comp.*, 58:369–380, 1992.

[SW12] Rahul Santhanam and Ryan Williams. Uniform circuits, lower bounds, and qbf algorithms. *Electronic Colloquium on Computational Complexity (ECCC)*, 19:59, 2012.

[Val77] Leslie G. Valiant. Graph-theoretic arguments in low-level complexity. In *6th Symposium on Mathematical Foundations of Computer Science*, volume 53 of *Lecture Notes in Computer Science*, pages 162–176. Springer, 1977.

[vM06] Dieter van Melkebeek. A survey of lower bounds for satisfiability and related problems. *Foundations and Trends in Theoretical Computer Science*, 2(3):197–303, 2006.

[VN12] Emanuele Viola and NEU. From RAM to SAT. Available at http://www.ccs.neu.edu/home/viola/, 2012.

[Vol99] Heribert Vollmer. *Introduction to circuit complexity.* Springer-Verlag, Berlin, 1999.

[Wil10] Ryan Williams. Improving exhaustive search implies superpolynomial lower bounds. In *42nd ACM Symp. on the Theory of Computing (STOC)*, pages 231–240, 2010.

[Wil11a] Ryan Williams. Guest column: a casual tour around a circuit complexity bound. *SIGACT News*, 42(3):54–76, 2011.

[Wil11b] Ryan Williams. Non-uniform ACC lower bounds. In *IEEE Conf. on Computational Complexity (CCC)*, 2011.

[Wil13] Ryan Williams. Natural proofs versus derandomization. In *ACM Symp. on the Theory of Computing (STOC)*, 2013.

[Zák83] Stanislav Zák. A turing machine time hierarchy. *Theoretical Computer Science*, 26:327–333, 1983.

ECCC ISSN 1433-8092

http://eccc.hpi-web.de